

Swedish to You is like Greek to Me

by Jon Wätte
h+@nada.kth.se

Abstract:

Reaching a global audience is good, both for your ego and your wallet. However, properly taking care of customers outside your own country requires some thought. Here is a summary of some of the more important issues. The focus is on how to resolve these issues in applications running under the Mac OS.

Going the international route doesn't mean a lot of work, it just means re-learning (which is worse is a matter of personal opinion.) Generally speaking, building an application that in itself is world-savvy is the right thing to do; calling the toolbox for comparing strings and finding word breaks builds a very strong foundation. Separating strings and icons so they can be translated into other languages is just the icing on the cake.

Below, I will repeatedly make references to "the US" and "Sweden" — you can substitute "the developer's native country" and "the localization target country" if you wish. The impact remains the same.

To Market!

If you know you're interested in a particular country, you should look over your sales organization. Do you have any contacts in the new country? Do you know the market structure, how to price your product, and how to support it?

If you're part of a larger company, chances are you do. If not, this is the time to start looking for a business partner in the country. Reading trade magazines from the country in question is a good way, if you understand the language. Otherwise you can look in the yellow pages for the larger cities, or maybe ask someone you *do* know (Apple Computer, for instance) I've found that a smaller independent company with a track record of several years in the business often makes for the best partner; larger corporations usually have too rigid structures and are less willing to adapt to your way of working.

If you're daring, you may want to start your own subsidiary in the new country (or maybe have it serve a whole region) but the scope of this paper does not cover that process. Suffice to say that it requires lawyers, even if the climate abroad isn't as lawsuit happy as in the US. (In Sweden, suing someone is one of the worst insults

you can imagine, no matter who wins.)

Another problem is the number of foreign countries you will have to cover. You can't serve "Europe" using one or two offices, since it's a region consisting of some twenty countries, with as many languages and cultures, and calling from one European country to another is expensive for the customer — using the equivalent of 800 numbers may help, but it's expensive for you.

Other issues to consider are cultural issues and the way people work — for instance, newspapers are planned and made differently in different countries. If you do not have area knowledge of the market you're trying to sell into, you have to have a business partner who does. Even if you plan on starting your own subsidiary in the end, hiring a professional localizer is necessary; preferably one with knowledge of the particular area your application focuses on. Again, trade press, or contacts already in the country, will help. You can also check out the Macintosh Services Directory.

The product

Apart from marketing, selling and supporting the product (which you can hand over to the natives) you should start working on your application. It needs to be localized, together with the manuals, packaging, and all other material relating to it. "Localization" here refers to the entire concept, not just translation (which is only part of localization, and actually not as important as you may think) My view is that support and manuals should go first, then packaging and sales, and last the actual application (provided the non-translated application still works in the new country.) One obvious reason why it's seldom done this way, is because you'll need screen

dumps for the manuals, and usually don't want to document an application with display in another language than the documentation.

Perhaps the main issue is: will this product sell abroad? Some products, like tax programs or accounting, need substantial localization before they work abroad. Others, like cheque-writing programs, probably won't sell no matter how much you try. (Bills are paid through a streamlined bank transfer system, not through mailed cheques in many parts of the world)

Another thing that varies are keyboard layouts (how many times have you written "option-8" when you mean "bullet character"? Well, for me, option-q means "bullet character".) Keyboard layouts also come into effect when menu shortcuts (command keys) are used, and as dead keys (The US keymap says option-u means add diacesis to the next character, while the Swedish keyboard has a specific key for that purpose, as well as having the letter ä on a key of its own) Also, assume that most non-alphabet characters (including numbers!) can be hard to get at; \ is option-shift-7 on a Swedish keyboard (and there's nothing on the key saying it is so).

Also beware of different ways of writing addresses and telephone numbers. How many digits in a telephone number? This may vary within one area code! Stockholm (area code 8 in Sweden) has numbers from 5 to 8 digits long. Other parts can have 4-digit area codes and still have both 5 and 6 digit numbers. How many digits in area codes? Again, this may vary. How are area codes marked? In some countries, it is put in parentheses. In others, it's followed by a dash. Still others use a slash, or just let the placement as the first group show its status. Some countries do not have area codes. Address registers should not assume a City-Zip-State field format; in Sweden you write Zip-City, and the British have a format with several cities as well as two zip codes.

So how do you know your localizer did a good job on your application? Some people just do not know how to spell, but still work in journalism or publishing. An impressive portfolio is not enough, unless it also mentions what users think of the results. You could enlist local user groups or trade mags (under non-disclosure) to help you find those last wrinkles. Local colleges or other institutes may also help. If you're completely lost, try talking to a trade & commerce representative at the nation's embassy.

Another problem is a matter of style; if you add jokes in the US version that *you* think are in taste in English, anything could happen in the localized version. The joke could disappear; it could be translated literally (which probably isn't such a great idea) or the localizer's favourite native joke could be substituted (be sure to look for "good sense of humour" on his resume...) Adopting a

slightly formal and conservative attitude in your application from the beginning usually works best and gives you the least surprises, especially if you get the message across to the localizer (if he's an easygoing type, stress formality, if he's a stout person, try to lighten it up.)

First stop Europe

For several reasons, Europe is among the first foreign markets to consider. However, it is essential to know that Europe is *not* as homogenous as the US; it's a veritable mess of countries, languages and cultures. Selling into Great Britain would seem a logical first step for a US software author, since the language is very similar, but the culture isn't the same, and for the best results, you have to localize for British English as well.

However, many things make localizing for western Europe easy:

- Most languages use the Roman script system
- Those that don't, still use one-byte scripts
- Business practices have common ground with the US.
- The infrastructure is in place; you can find suitable business partners.

The main issues here are to separate text strings for easy translation, document how string concatenation is done so the localizer has a fighting chance of making sense of constructed messages, use the correct date, time and currency symbols, sort in the right order, and design screen layouts so there is space for the translated text.

Fortunately, on the Macintosh, this is easy, since many of these functions have been available since System 1.0, and the rest came in with System 4.2. Putting strings in resources instead of literals might require some

forethought, but is not that hard to do right from the beginning (if you haven't, a tool for you will be presented below)

Documenting string concatenation is a little harder, but again, if you thought about it from the beginning, it's much easier to do. An example of this is the common error dialog "Could not complete the last command, because " which will be concatenated with an appropriate error code string, such as "there is not enough memory." or "a required resource is missing." or "an error of type -35 occurred."

However, when translating these messages, it might be impossible to make them sound natural in the target language. Some foresight, again, will save the day. The main trick here is adding flexibility by concatenating empty strings; instead of just having the string "Could not..." and the error string, you should concatenate the string "Could not..." with the error message and then with a specific, unique empty string. This will let the localizer add any necessary trailer phrase, without adding it to every error message string. Best of all, however, is designing so you do not have to concatenate strings. Users of `sprintf()` beware!

On a tangential point, usage of signed chars for strings is another pitfall, since most character codes are defined on the range 0–255, not -128–127. Similarly, it is not prudent to check if characters are < 32 (space) to check for control characters if you use signed chars. Make it a habit to always use unsigned chars for string data.

Getting date and time and currency formats right is a special case of concatenating strings; however, these are handled by calls in the Mac OS part called the International Utilities. Here, you need to throw your prejudices aside: you'll have to assume that the OS is written in the most efficient manner possible, and that you can't do any better yourself, else you will start running into problems down the line. Localization is very much an "all or nothing" process; users perceive a badly localized product as worse than one that is not localized at all.

Set the Font

It is important that a valid (and preferably current) font is set in the currently active GrafPort (which also should be valid) since the International Utilities and Script Manager use this information to find out what script, language and region you're looking at right now.

Use the built-in functions `IUDateString` and `IUTimeString` for date and time. Use `IUGetIntl(0)` to access a handle specifying how you should format currency information. Use `IUGetIntl(1)` to get a list of day and month names, as well as information on how `IUDateString` and `IUTimeString` behave.

Swedish to You is like Greek to Me

Is your program metric? If it assumes any particular measurement system, it cuts itself out from large parts of the world. However, the metric system is an accepted world standard, and you can promote reason by not supporting any other measurement system. Meters, Newton and litres fit much better together than furlongs and footlamberts.

Finding word breaks is a little harder; you can't assume you know what characters are considered letters and which are not. Easiest is to use `FindWord()` with a break table of `NULL` (for word selection) or `-1` (for word wrapping) This comes into play when you add functionality to `TextEdit` fields for next-word and previous-word movement, as well as when you parse input from users.

Scripting

Many applications have their own script language. However, the grammar and structure of natural languages wary wildly; in German you put all verbs except one at the end of a sentence, while in Japanese you toss them into a jar, shake, and see what comes out... (OK, so I'm a little prejudiced myself!)

Not only do you have to consider token order, you also have to consider token recognition. Probably the best thing would be to keep a list of tokens in a `STR#` list, and hash them when the application starts up. Then use `FindWord` to tokenize your script.

If you want case insensitivity, you will have to call `LwrString` and `UprString`, or maybe `Transliterate`, before hashing. This also goes for specific lower-case and upper-case functions. You can most definitely NOT trust the sixth bit of a byte to have any significance on case!

For string comparisons, use `IUCompString`. Even if sorting is a major part of your application, and you think you lose some performance by calling an A-trap, you should use `IUCompString`, since you will lose even more customers when your application doesn't sort right. As an example, in Swedish you would sort é with e, ü with y while å, ä and ö are completely separate letters of their own, sorted at the end of the alphabet. (No, there is no law saying the alphabet should only have 26 characters!)

Abcdefghijklmnopqrstuvwxyzåäö

The Swedish alphabet

Displaying numbers is equally perilous; some countries use thousands separators, others use a comma for decimals, still others do not print the 0 in fractions less than one. Luckily, there are calls to the Macintosh OS for this purpose, too; `FormatX2Str()` converts a floating-point extended value to a string according to a format that you can store as a string and compile at program start-up using `Str2Format()`.

Graphics

Not all icons are as natural as you may think; in the US, a check mark is used to indicate something as “checked” or “OK,” while in Sweden it's the most feared of marks for school children, indicating errors. Where an American spreadsheet program uses X and √ for cancel and accept, the same program in Sweden could use √ and O. Better yet would be a trash can and an enter arrow (down-left arrow)

Other cultures may have prejudices against other kinds of symbols; the most well-known of these might be the international red cross, which uses a red crescent in the Islamic parts of the world.

You should also make plenty of room in your dialogs; do not assume you know where text goes in windows and what width is

Swedish to You is like Greek to Me

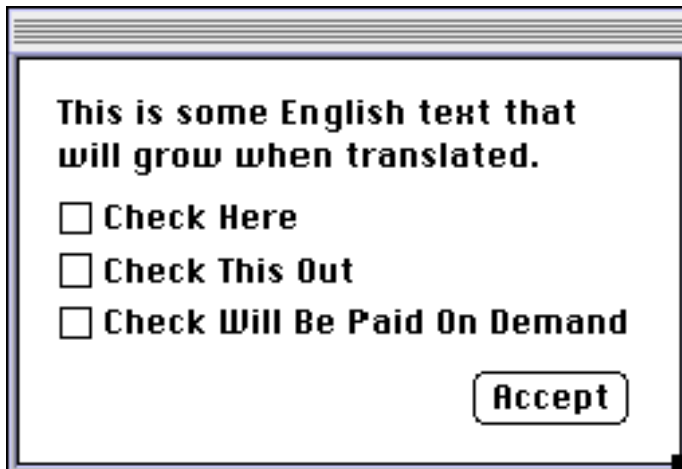
needed to fit it in. Usually text will not grow more than 20% when being translated from English to Swedish or German, but in some cases you simply *have* to expand text on two lines to four or five lines. Leave *lots* of space in text fields; similarly, make buttons wider than you would think reasonable (80 pixels is a good minimum general width) Do not put entire phrases in buttons.

If you design your own fonts, make sure to include the entire character set in the font, not just A-Z and 0-9. Also make sure that you leave enough leading, ascent and descent space in the font, the letter Å should look like the letter A with a small ring above it (not touching) and still not catch with the lower parts of the letters p or g. (Note that the font this paper is printed in, Palatino, does *not* follow these conventions, which makes text in Swedish set in Palatino hard to read.)

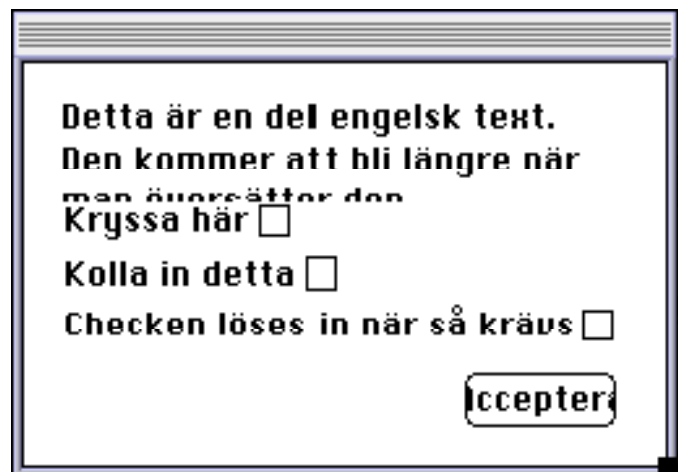
Next stop the World

When you start getting calls from Japan or Korea, you know it's time to start looking not only at the International Utilities pages of Inside Macintosh, but also at the Script Manager. Earlier, the Macintosh could use only one script system at a time, selected by which System file you booted with. With the introduction of System 7.1, that is no longer true; the user can switch scripts right in the middle of doing something, and your application must be prepared to handle that.

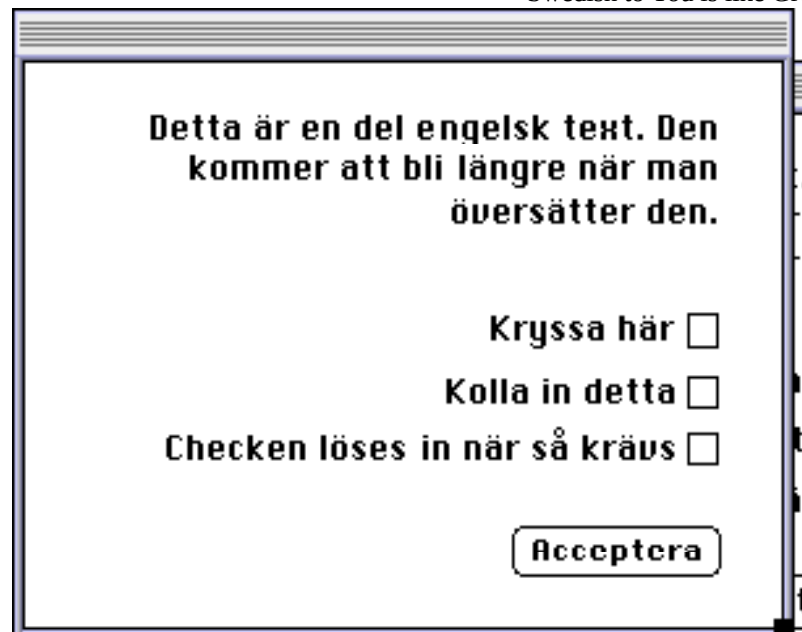
The first and most noticeable difference is scripts that go from right to left. This throws out the normal concept of “left justification” meaning text starts at the left side of the screen; instead you have a “system justification” replacing “left justification” and a new kind of justification meaning “force left.” Dialogs will have all check boxes and radio buttons reversed, i.e. the text comes to the right of the symbol, and they are right justified within the bounding box of the control. For this reason, you should design your dialogs so that all controls line up both in the right and left edges.



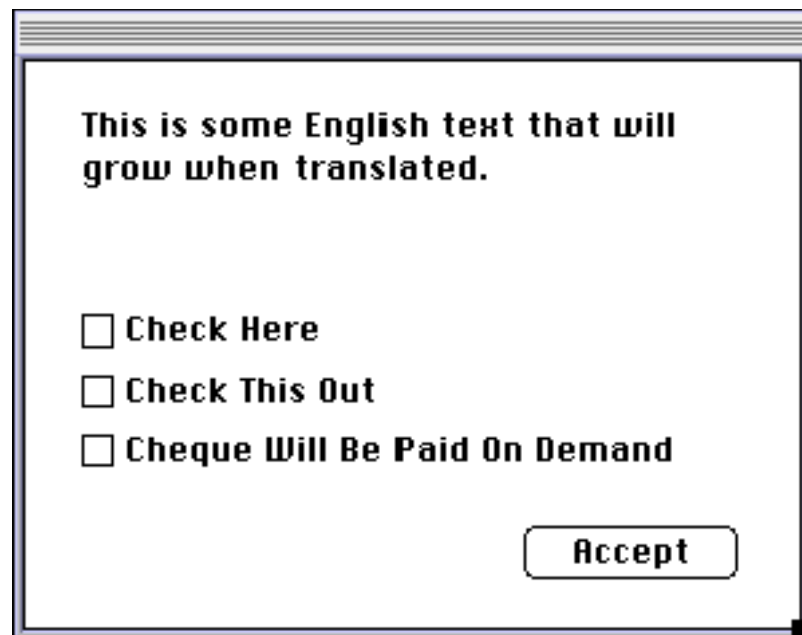
How not to design a dialog



Why you shouldn't (Swedish is not a right-to-left script, though)



This is how it should look



So you'll have to design it like this

The second important difference is multi-byte scripts. Since the number of glyphs is so large, one byte is not enough to represent all glyphs; instead two or more bytes are used to represent one glyph or “letter.” The implications of this are that you no longer can assume one byte == one letter. Instead, you will have to call CharByte to determine whether the byte you’re looking at is the first, middle, last or only byte of a character. When doing so, make sure that the current font and port are set accordingly. You can

view the script of some text to be just another attribute such as font or style; indeed, the font of a text determines its script.

When examining text, you can no longer use a pointer to the text and pass a 0 for the “offset” parameter many toolbox calls want; since the meaning of bytes may change with context, you need to point to the start of the text buffer (or other known break point, such as right after a carriage return) and supply the appropriate offset into the text, else calls to the Script Manager will not work correctly.

The script manager also has code for handling hit detection, selections across script boundaries, and other generally unpleasant artefacts of mixed scripts; however, if at all possible, you should let TextEdit do the dirty work for you. There are also some commercial word-processing libraries on the market handling multi-script text. The Word Solutions Engine from DataPak software is one example (the author is not affiliated with DataPak in any other way than as a customer).

Collecting the Dough

So, are you ready to take on the world? Only your users will know; when you move outside your own experience area into foreign languages, you will have to trust other people with adapting your product to local conditions. The best way of knowing that they do a good job, is letting them in on a share of the profits from that market. However, since more people are involved, there will be a lesser share for you, which may tempt you to raise the price for the localized version.

Don't.

Users of today are generally price-conscious, and while you may see the cost of the localization process as something particular to that country, your users will wonder why your software costs \$800 at their local shop, while they can buy it for \$195, shipping and handling included, mail-order from the US. Then they'll call you in the US, speaking French and wonder why it doesn't work correctly, and you're back at square one. Similarly, forcing the localized version to run with a specific system software language (like AutoDoubler) or keyboard (like Quark XPress) generates much more bad karma than good profits.

If that is the case, you're probably better off not localizing at all, but instead selling the US version abroad, too — only, since you now use the Toolbox for all critical tasks, chances are that the functions that are important (sorting data, printing dates, etc.) work for those users, as well, even if it's called something obscure such as “Paste Date” instead of “Klistra in datum” in the edit menu.

One way to avoid the large price differences is to recognize the benefits US customers get (after all, there are lots of Mandarin writing or Spanish talking persons living and working in the US)

and spread the price for the extra work needed to convert a non-localizable application to a localizable one evenly across all markets.

Tools

So how do you localize your software? Well, you could conceivably have your localizers bash away at your program with ResEdit; however, that is a less than ideal solution. Apple has a tool called AppleGlot which is ideally suited for translating resources; it handles most resource types containing strings, and lets you type in translations using any word processor or text editor. You can build dictionaries of translated strings to re-use for the next version, or even for another application.

However, running AppleGlot requires your strings and other localizable items to already reside in resources; for an application with lots of hard-coded string constants, getting there can be quite a feat. Enter CStringExtractor, a simple utility (in itself not localizable at all) which extract string literals from C source code, and inserts calls to a string literal handling library instead.

Usage is simple: Run the utility, enter a starting resource ID, and select a folder containing C source files. All C or pascal string literals will be replaced with calls to the functions CLiteral() and PLiteral(), and a source file will be created containing all the extracted literals. Then run the string compiler utility to collect all separated strings into a resource file that you should include in your program.

NOTE: the string extractor makes some assumptions about how strings are used in code, and will not handle pre-processor symbols. You should only attempt to do this on a copy of your source code. Full source code is included, so you can make whatever changes you need to the process. Please send me a line if you find any bugs or make any improvements.

The string extractor and compiler code is in itself good examples of how *not* to write internationalizable code; it assumes a one-byte script.

What now?

If you're developing for the Macintosh, I suggest you read the Script Manager chapter of Inside Mac V. It will give you a good idea of generally what level of support is available. Then use that support. Don't assume dates are written YY-MM-DD, don't assume one byte == one character; don't even assume strcmp will do a good job. The toolbox may well be more efficient, and it will do the job correctly.

If you're a software designer, or user interface designer, you need

Swedish to You is like Greek to Me

to take localization into account at an early stage. Make provisions for easy translation of menus and message strings. Don't assume that a certain message takes a certain finite amount of memory; don't assume it will fit in a tight space on the screen. Don't rely on a certain graphical language or culturally specific way of working. Design for flexibility.

Open your mind to foreign cultures. Who knows, maybe it will happen that you possibly *have* to make a two-week trip to Cannes to study European bea... I mean ways of working.

I may be reached as h+@nada.kth.se if you're interested in discussing the contents of this paper, or find errors or omissions in it.